

Processors and Instructions

Todor Stoyanov

Department of Computing and Software
McMaster University

26.09.2022

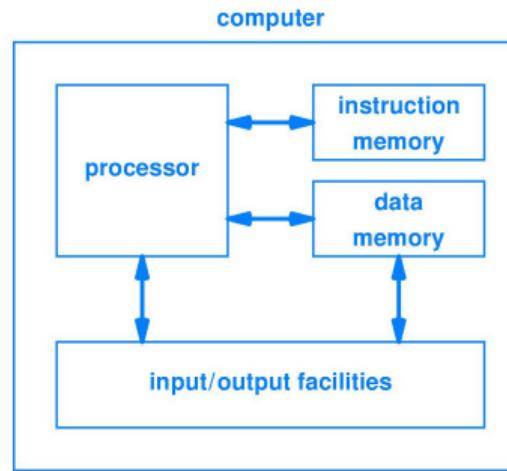
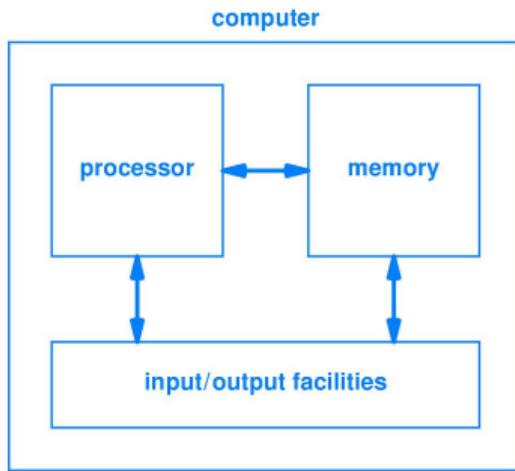




Outline

1. Computer Architectures
2. Processor Types and Structure
3. Instruction Set Architecture
4. Pipelined Processors

Harvard and Von Neumann Architectures



Von Neumann Architecture

- Flexibility
- Memory bottleneck

Harvard Architecture

- Less susceptible to bottleneck
- Security advantage
- Rigid split



Processor Types and Categories

Processor

A digital device that can perform multi-step computation.

Types

- Fixed logic
- Selectable logic
- Parametrizable logic
- Programmable logic



Processor Types and Categories

Processor

A digital device that can perform multi-step computation.

Types

- Fixed logic:

Function fixed in hardware.

- Selectable logic:

Choose one of several fixed functions.

- Parametrizable logic:

Parameters govern computation.

- Programmable logic:

List of instructions provided at runtime.



Processor Types and Categories

Processor

A digital device that can perform multi-step computation.

Types

- Fixed logic
- Selectable logic
- Parametrizable logic
- Programmable logic

Categories

- Co-processors
- Microcontrollers
- Embedded systems
- General purpose



Processor Types and Categories

Processor

A digital device that can perform multi-step computation.

Types

- Fixed logic
- Selectable logic
- Parametrizable logic
- Programmable logic

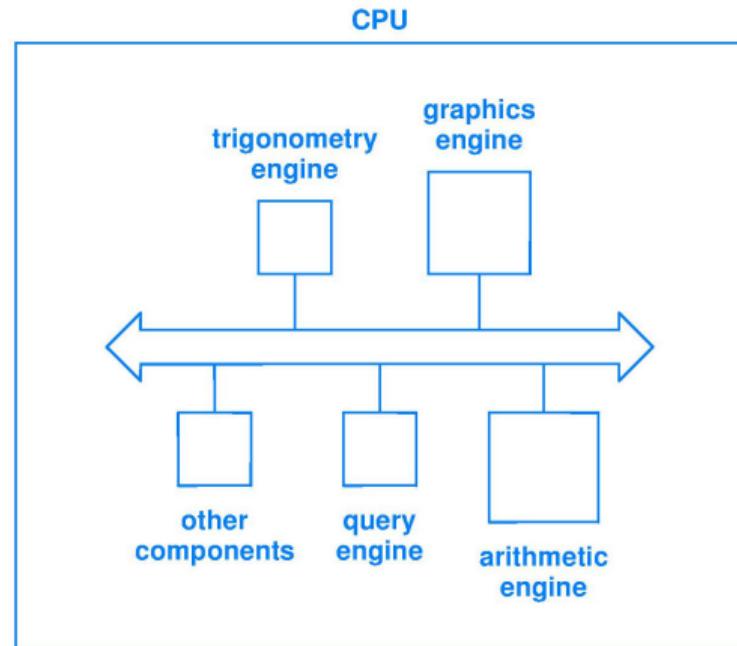
Categories

- Co-processors:
Dedicated function, fixed/selectable logic.
- Microcontrollers:
Programmable logic, direct hardware control.
- Embedded systems:
Real-time OS, dedicated hardware.
- General purpose:
Interchangeable and compatible for multiple systems.

Processor Structure



- ALU
- Local storage
- Controller
- External interface
- Internal connections





Processors



Or www.menti.com and enter number **7823 7075**



Instructing Processors

- Stored program processors
- Sequence of instructions stored in memory
- Processor conceptually separate from program and function
- Allows for abstraction



The Fetch-Execute Cycle

```
ip = start of program  
Repeat forever  
    instruction = fetch (memory [ip])  
    execute instruction  
    ip++
```

The Fetch-Execute Cycle

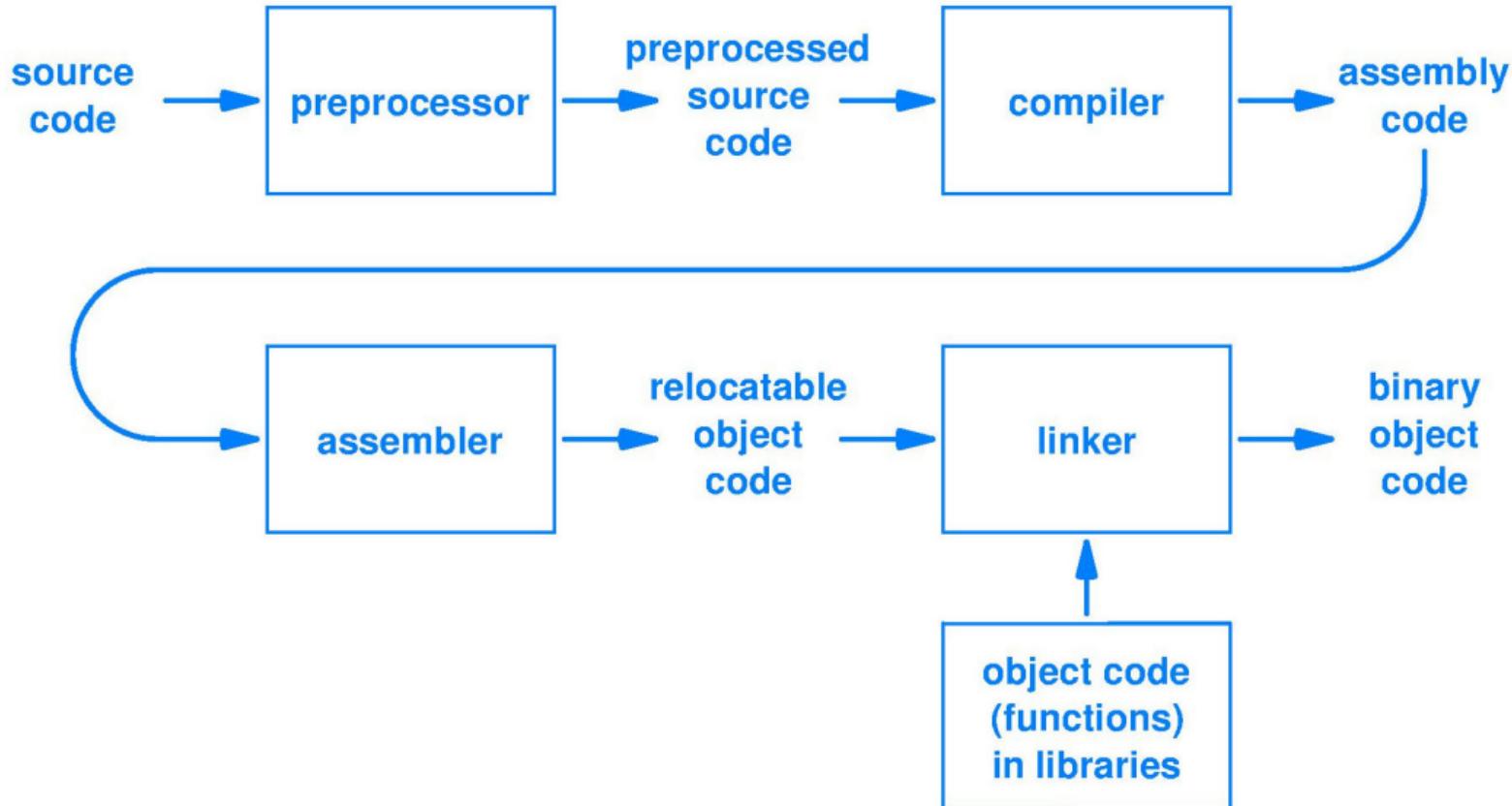


```
ip = start of program
Repeat forever
    instruction = fetch (memory [ip])
    execute instruction
    ip++
```

Open questions:

- How are instructions represented?
- How do we know where to start?
- How do we move to the next instruction?
- What happens when we reach the end?

High-level Programming





High-level Programming: Example

```
#include<stdio.h> // for printf
#include<stdlib.h> // for atof
#include<math.h> // for fabsf

static float sqrarg;
#define SQR(a) ((sqrarg=(a)) == 0.0 ? 0.0 : sqrarg*sqrarg)

float FastInvSqrt(float x) {
    float xhalf = 0.5f * x;
    int i = *(int*)&x;
    i = 0x5f3759df - (i >> 1);
    x = *(float*)&i;
    x = x*(1.5f-(xhalf*x*x));
    return x;
}

int main(int argc, char* argv[]) {
    if (argc!=2) {
        printf("usage: %s float_arg\r\n", argv[0]);
        return -1;
    }

    float test_number = atof(argv[1]);
    float err = fabsf(1.0f/test_number - FastInvSqrt(SQR(test_number)));

    printf("Approximation error is %05f\r\n", err);

    return 0;
}
```

High-level Programming: Example



```
.file  "fastinv.c"
.text
.p2align 4
.globl FastInvSqrt
.type  FastInvSqrt, @function
FastInvSqrt:
.LFB39:
    .cfi_startproc
    endbr64
    movd  %xmm0, %edx
    movl  $1597463007, %eax
    movss .LC1(%rip), %xmm1
    mulss .LC0(%rip), %xmm0
    sarl  %edx
    subl  %edx, %eax
    movd  %eax, %xmm2
    mulss %xmm2, %xmm0
    mulss %xmm2, %xmm0
    subss %xmm0, %xmm1
    movaps %xmm1, %xmm0
    mulss %xmm2, %xmm0
    ret
    .cfi_endproc
.LFE39:
    .size  FastInvSqrt, ..-FastInvSqrt
    .section      .rodata.str1.1,"aMS",@progbits,1
.LC3:
    .string "usage: %s float_arg\r\n"
.LC6:
    .string "Approximation error is %05f\r\n"
    .section      .text.startup,"ax",@progbits
    .p2align 4
    .globl main
    .type  main, @function
main:
```

High-level Programming: Example

```
tsv@tsv-laptop:~$ hexdump fastinv.o
00000000 457f 464c 0102 0001 0000 0000 0000 0000
00000010 0003 003e 0001 0000 1140 0000 0000 0000
00000020 0040 0000 0000 0000 39d0 0000 0000 0000
00000030 0000 0000 0040 0038 000d 0040 001f 001e
00000040 0006 0000 0004 0000 0040 0000 0000 0000
00000050 0040 0000 0000 0000 0040 0000 0000 0000
00000060 02d8 0000 0000 0000 02d8 0000 0000 0000
00000070 0008 0000 0000 0000 0003 0000 0004 0000
00000080 0318 0000 0000 0000 0318 0000 0000 0000
00000090 0318 0000 0000 0000 001c 0000 0000 0000
000000a0 001c 0000 0000 0000 0001 0000 0000 0000
000000b0 0001 0000 0004 0000 0000 0000 0000 0000
000000c0 0000 0000 0000 0000 0000 0000 0000 0000
000000d0 0658 0000 0000 0000 0658 0000 0000 0000
000000e0 1000 0000 0000 0000 0001 0000 0005 0000
000000f0 1000 0000 0000 0000 1000 0000 0000 0000
00000100 1000 0000 0000 0000 02f5 0000 0000 0000
00000110 02f5 0000 0000 0000 1000 0000 0000 0000
00000120 0001 0000 0004 0000 2000 0000 0000 0000
00000130 2000 0000 0000 0000 2000 0000 0000 0000
00000140 01c8 0000 0000 0000 01c8 0000 0000 0000
00000150 1000 0000 0000 0000 0001 0000 0006 0000
00000160 2db0 0000 0000 0000 3db0 0000 0000 0000
00000170 3db0 0000 0000 0000 0260 0000 0000 0000
00000180 0268 0000 0000 0000 1000 0000 0000 0000
00000190 0002 0000 0006 0000 2dc0 0000 0000 0000
000001a0 3dc0 0000 0000 0000 3dc0 0000 0000 0000
000001b0 01f0 0000 0000 0000 01f0 0000 0000 0000
000001c0 0008 0000 0000 0004 0000 0004 0000 0000
000001d0 0338 0000 0000 0000 0338 0000 0000 0000
000001e0 0338 0000 0000 0000 0020 0000 0000 0000
000001f0 0020 0000 0000 0000 0008 0000 0000 0000
```

High-level Programming: Example

```
tsv@tsv-laptop:~$ nm fastinv.o
0000000000004010 B __bss_start
0000000000004010 b completed.8061
    w __cxa_finalize@@GLIBC_2.2.5
0000000000004000 D __data_start
0000000000004000 W data_start
0000000000001170 t deregister_tm_clones
00000000000011e0 t __do_global_dtors_aux
000000000003db8 d __do_global_dtors_aux_fini_array_entry
0000000000004008 D __dso_handle
0000000000003dc0 d _DYNAMIC
0000000000004010 D _edata
0000000000004018 B _end
0000000000001230 T FastInvsqrt
00000000000012e8 T _fini
0000000000001220 t frame_dummy
000000000003db0 d __frame_dummy_init_array_entry
00000000000021c4 r __FRAME_END__
0000000000003fb0 d __GLOBAL_OFFSET_TABLE__
    w __gmon_start__
0000000000002060 r __GNU_EH_FRAME_HDR
0000000000001000 t _init
0000000000003db8 d __init_array_end
0000000000003db0 d __init_array_start
0000000000002000 R _IO_stdin_used
    w __ITM_deregisterTMCcloneTable
    w __ITM_registerTMCcloneTable
00000000000012e0 T __libc_csu_fini
0000000000001270 T __libc_csu_init
    U __libc_start_main@@GLIBC_2.2.5
0000000000001080 T main
    U __printf_chk@@GLIBC_2.3.4
00000000000011a0 t register_tm_clones
0000000000001140 T _start
    U strtod@@GLIBC_2.2.5
0000000000004010 D __TMC_END__
```



Instruction Set Architecture

- The design of all available instructions for a processor is known as the Instruction Set Architecture (ISA)
- Defines
 - What instructions are available.
 - What each instruction does. Pre/post conditions.
 - Number of operands, operand encoding, instruction encoding, result
- Three most-common ISA families today: x86, arm and RISC-V.



Assembly Languages

- Low-level programming language
- An intermediate step to machine instructions
- Direct mapping
- Human readable, some shortcuts, macros

```
.file  "fastinv.c"
.text
.p2align 4
.globl FastInvSqrt
.type  FastInvSqrt, @function
FastInvSqrt:
.LFB39:
    .cfi_startproc
endbr64
    movd    %xmm0, %edx
    movl    $1597463007, %eax
    movss   .LC1(%rip), %xmm1
    mulss   .LC0(%rip), %xmm0
    sarl    %edx
    subl    %edx, %eax
    movd    %eax, %xmm2
    mulss   %xmm2, %xmm0
    mulss   %xmm2, %xmm0
    subss   %xmm0, %xmm1
    movaps  %xmm1, %xmm0
    mulss   %xmm2, %xmm0
    ret
    .cfi_endproc
.LFE39:
    .size   FastInvSqrt, .-FastInvSqrt
    .section        .rodata.str1.1,"aMS",@progbits,1
.LC3:
    .string "usage: %s float_arg\r\n"
.LC6:
    .string "Approximation error is %05f\r\n"
    .section        .text.startup,"ax",@progbits
    .p2align 4
.globl main
```



Types of Instructions

- Typical structure of an instruction:

- Different types of instructions:

- ALU operations
- Register access
- Memory access
- Program flow control



Branching

- Moving the instruction pointer to a different location in program.
- Necessary for conditionals, loops, etc.

```
ip = start of program
Repeat forever
    instruction = fetch (memory [ip])
    tmp = ip+1
    execute instruction
    ip = tmp
```

```
jump ADDR  tmp=ADDR
jump +X    tmp=ip+X
bne +X    if(cond) tmp=ip+X
```

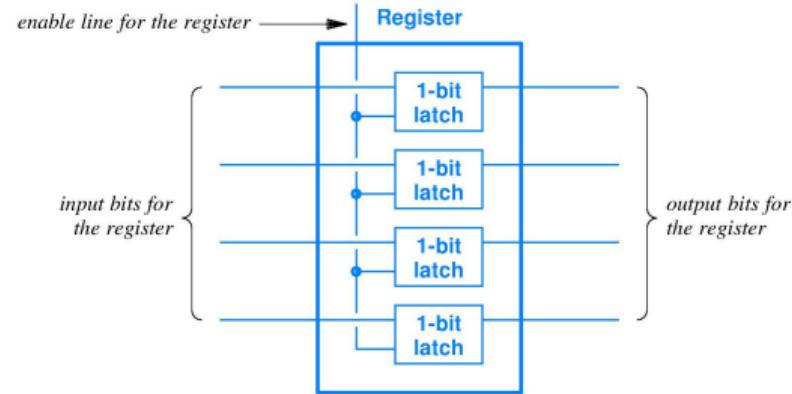


Example: Branching

```
#loop label
loop: add r1 r2      #increment counter
      memo r3 r7    #perform some operation
      cmp r2 r4      #compare to invariant
      bne loop       #conditional branch
#main label
main:                      #goes on
```

Registers

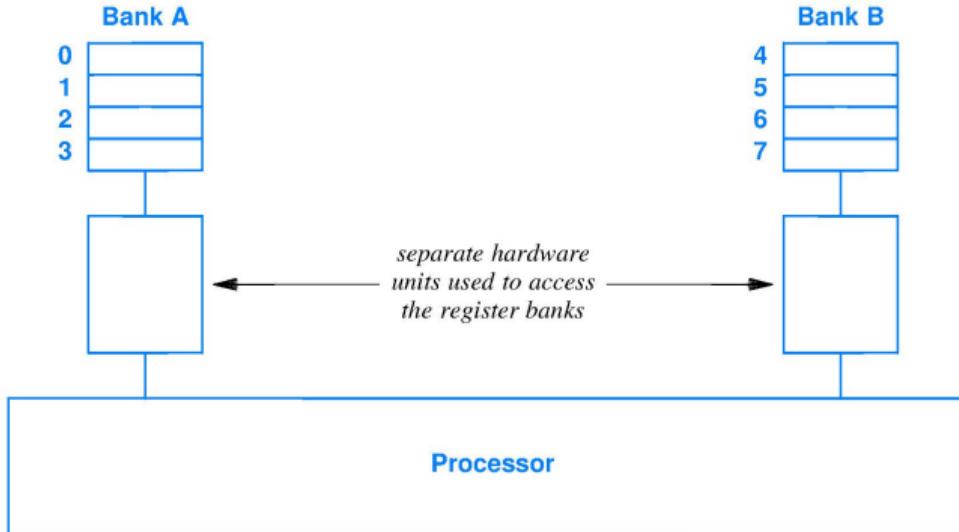
- fetch / store semantics
- Types:
 - general-purpose
 - floating point
 - instruction pointer
 - comparison operation



Typical workflow with registers

- Load data from memory to register.
- Perform ALU operation.
- Store result from register to memory.

Register Banks



- Allows parallel access within same clock cycle → efficiency.
- Some operations require operands on different banks.
- Register bank conflicts.

Example: Register Bank Conflict

Example: Two banks. r0 to r3 on bank A, r4 to r7 on Bank B.

```
#loop label
loop: add r1 r1 r2      #r1 = r1+r2
      add r4 r2 r3      #r4 = r2+r3
      memo r3 r3 r7     #memory operation
      cmp r2 r4          #is r2>r4?
      bne loop           #conditional branch
```



Example: Register Bank Conflict

Example: Two banks. r0 to r3 on bank A, r4 to r7 on Bank B.

```
#loop label
loop: add r1 r1 r2    #r1 = r1+r2
      add r4 r2 r3    #r4 = r2+r3
      memo r3 r3 r7   #memory operation
      cmp r2 r4        #is r2>r4?
      bne loop         #conditional branch
```

How do we fix this?

Two options:

- Re-assign register numbers
- Copy data

Example: Register Bank Conflict

Example: Two banks. r0 to r3 on bank A, r4 to r7 on Bank B.

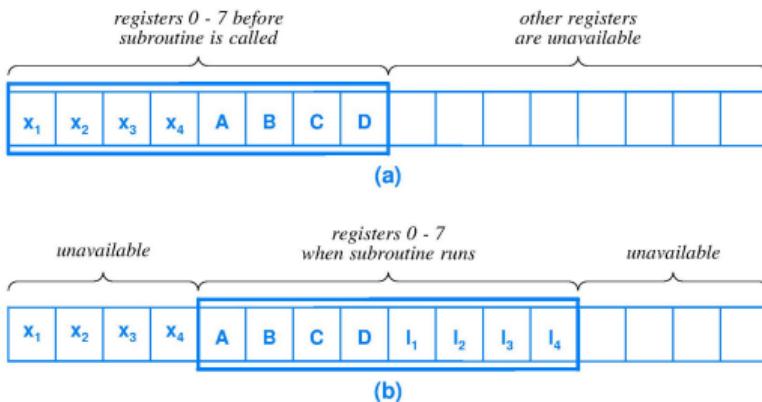
```
cpy r2 r5          #copy r2 to r5
#loop label
loop: add r1 r1 r5    #r1 = r1+r5
      add r4 r5 r3    #r4 = r5+r3
      memo r3 r3 r7    #memory operation
      cmp r2 r4         #is r2>r4?
      bne loop          #conditional branch
```

Now you



Or www.menti.com and enter number **7823 7075**

Subroutines and Register Windows



- Subroutine in assembly like a function in C.
- How do we pass arguments?
- Useful for system functions.



How do we decide what instructions to provide?

- Each instruction performs a complex operation.
- Instructions take variable number of clock cycles.
- Fewer instruction calls.
- Each instruction performs simple operation.
- Instructions all take same number of clock cycles.
- Many instruction calls.

Complex Instruction Set Computer (CISC)

Reduced Instruction Set Computer (RISC)



How do we decide what instructions to provide?

- Each instruction performs a complex operation.
- Instructions take variable number of clock cycles.
- Fewer instruction calls.
- Each instruction performs simple operation.
- Instructions all take same number of clock cycles.
- Many instruction calls.

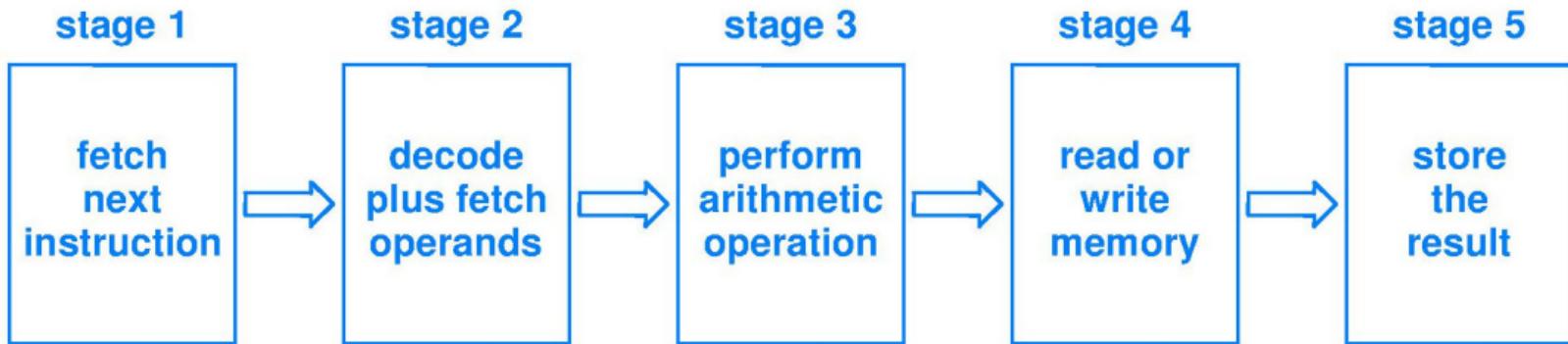
Complex Instruction Set Computer (CISC)

Reduced Instruction Set Computer (RISC)

- Computational architecture, improved throughput.
- Conditions to use a pipeline computational architecture:
 - Computation can be divided in distinct stages.
 - Each stage takes similar time to complete.
 - It is efficient to move data between stages.

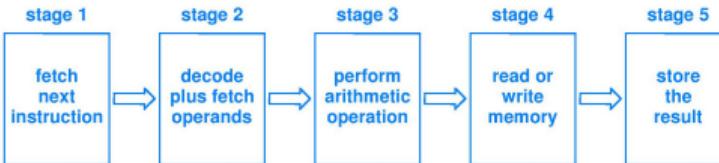


RISC Pipeline



Example: Pipelining

```
1:loop: add r1 r1 r2
2:      add r5 r2 r3
3:      add r4 r2 r3
4:      add r5 r0 r3
5:      sub r7 r1 r3
6:      cmp r5 r4
7:      bne loop
```

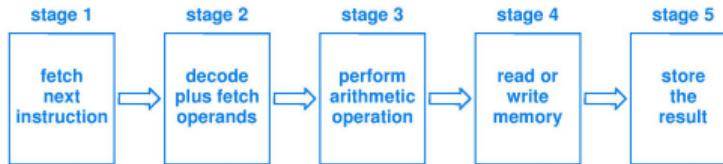


stp	stg1	stg2	stg3	stg4	stg5	done
0						
1						
2						
3						
4						
5						
6						
7						
8						
9						

Example: Pipelining

```

1:loop: add r1 r1 r2
2:      add r5 r2 r3
3:      add r4 r2 r3
4:      add r5 r0 r3
5:      sub r7 r1 r3
6:      cmp r5 r4
7:      bne loop
  
```



stp	stg1	stg2	stg3	stg4	stg5	done
0	1					
1	2	1				
2	3	2	1			
3	4	3	2	1		
4	5	4	3	2	1	
5	6	5	4	3	2	1
6	7	6	5	4	3	2
7	7	6		5	4	3
8		7	6		5	4
9		7		6		5



Pipeline Stalls

- Pipelines produce one output instruction per cycle.
- *but* a stall disrupts the flow.
- A “bubble” that floats until pipeline re-starts
- Stalls are also called *hazards*.



Dealing with Hazards

- Data hazards — waiting for data from earlier instruction.
- Control hazards — incorrect instruction is in the pipeline (branching).
- Structural Hazards — resource conflict.



Dealing with Hazards

- Data hazards — waiting for data from earlier instruction.

Data forwarding between stages. Re-arrange computation.

- Control hazards — incorrect instruction is in the pipeline (branching).

Conditional branch prediction. If prediction wrong, flush pipeline.

- Structural Hazards — resource conflict.

Load data in parallel, e.g. using banks

Example: Document Stalls

```
imd    r0(0x01)      # set immediate value
imd    r1(0x04)      # set immediate value
imd    r2(0x00)      # set immediate value
imd    r3(0x0F)      # set immediate value
load   r3(0x0A),r4  # load data in r4
label1: bge   r1,r2,label2 # branch if r1>=r2
        add   r0,r4,r4    # r4=r0+r4
        add   r0,r2,r2    # r2=r0+r2
        jmp   label1      # jump to label1
label2: store r3,r4    # memory[ r3 ]=r4
```

Example: Document Stalls

```
imd    r0(0x01)      # set immediate value
imd    r1(0x04)      # set immediate value
imd    r2(0x00)      # set immediate value
imd    r3(0x0F)      # set immediate value
#stall 4 cycles
load   r3(0x0A),r4  # load data in r4
#on 2nd run: stall 3 cycles
label1: bge   r1,r2,label2 # branch if r1>=r2
        add    r0,r4,r4      # r4=r0+r4
        add    r0,r2,r2      # r2=r0+r2
        jmp   label1       # jump to label1
label2: store r3,r4      # memory[ r3 ]=r4
```

Total stalls: $4 + 4 \times 3 = 15$

Now you



Or www.menti.com and enter number **7823 7075**



Example: Minimize Stalls

```
imd    r3(0x0F)      # set immediate value
imd    r1(0x04)      # set immediate value
imd    r2(0x00)      # set immediate value
imd    r0(0x01)      # set immediate value
#stall 1 cycle
load   r3(0x0A),r4  # load data in r4
#1st run 1 cycle; 2nd run 2 cycles
label1: bge   r1,r2,label2 # branch if r1>=r2
        add    r0,r2,r2      # r2=r0+r2
        add    r0,r4,r4      # r4=r0+r4
        jmp   label1       # jump to label1
label2: store r3,r4      # memory[ r3 ]=r4
```

Total stalls: $1 + 1 + 4 \times 2 = 10$



Summary

- Two design paradigms: separate instruction / data memory or combined.
 - Hybrids do exist!
- Structure of a processor
- Instructions and Instruction Set Architecture
- RISC and CISC
- Pipelines, pipeline stalls and mitigation measures



Survey



Or www.menti.com and enter number **7823 7075**